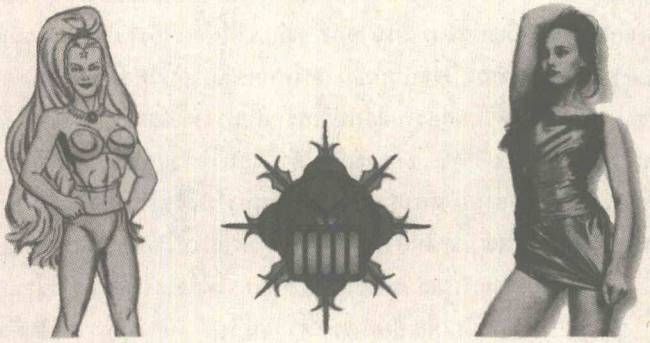
Unix

ไพโรจน์ ไววานิชกิจ

Shell Programming กับ UNIX ตอนที่ 1

เรื่องราวของ Shell Programming เพื่อเป็นประโยชน์ต่อผู้สนใจ โดยเน้นไปที่เชลล์แบบบอร์น และเชลล์แบบซี เป็นหลัก



ผู้เขียนเคยนำเสนอบทความชุด "ก้าวแรก... สู่โลก UNIX" ในวารสารไมโครคอมพิวเตอร์ โดยมีจุดประสงค์เพื่อแนะนำให้ผู้อ่านรู้จักกับระบบปฏิบัติการ UNIX ซึ่งทุกวันนี้ก็ได้ก้าวเข้ามาเป็นอีกระบบปฏิบัติการหนึ่งที่มีการใช้งานในเครื่องคอมพิวเตอร์ส่วนบุคคลพอสมควร หากเทียบกับระบบปฏิบัติการ DOS จะพบว่า UNIX มีขีดความสามารถในการอำนวยความสะดวกต่อผู้ใช้งานมากกว่าโดยเฉพาะในเรื่องของการเขียนโปรแกรมเชลล์หรือที่เรียกกันทั่วไปว่า Shell Programming ผู้เขียนถือเป็นโอกาสอันดีที่จะได้นำเสนอเรื่องราวของ Shell Programming เพื่อเป็นประโยชน์ต่อผู้สนใจ โดยบทความชุดนี้จะเน้นไปที่เชลล์แบบบอร์น (Bourne Shell) และเชลล์แบบซี (C Shell) เป็นหลัก

เชลล์คืออะไร

สำหรับผู้ที่ยังไม่ทราบถึงความหมายของเชลล์ (Shell) ผู้เขียนจะขออธิบายสั้น ๆ ว่า ในระบบปฏิบัติการ UNIX ภารกิจหรือกระบวนการต่าง ๆ ที่เกิดขึ้นในลักษณะปฏิสัมพันธ์ (Interactive) ระหว่างบรรดาผู้ใช้งานและตัวระบบปฏิบัติการจะเกิดขึ้นได้ด้วยการสนับสนุนของเชลล์ โดยเชลล์เป็นโปรแกรมซึ่งถูกโหลดเข้าสู่หน่วยความจำของเครื่องคอมพิวเตอร์ในขณะที่ทำการเปิดเครื่อง คอยทำหน้าที่รับคำสั่งจากผู้ใช้งานแล้วแปรความหมายพร้อมกับส่งต่อไปยังกระบวนการ หรือที่นิยมเรียกกันว่าโปรเซส (process) ต่าง ๆ เพื่อนำไปปฏิบัติการตามคำสั่งดังกล่าวนั้น อีกทั้งยังทำหน้าที่รับผลที่ได้จากการนำคำสั่งไปปฏิบัติเพื่อส่งกลับมาแจ้งต่อผู้ใช้งาน อนึ่ง คำว่าผู้ใช้งานนี้ในบางครั้งอาจไม่ได้หมายถึงตัวบุคคลที่ใช้เครื่องคอมพิวเตอร์เสมอไป แต่อาจหมายถึงโปรเซสต่าง ๆ ที่กำลังทำงานอยู่ในระบบปฏิบัติการ UNIX ก็ได้ ผู้เขียนจะขอกล่าวถึงเรื่องนี้โดยละเอียดในภายหลัง

ในโลกของระบบปฏิบัติการ UNIX มีเชลล์ที่ถูกสร้างขึ้นและได้รับความนิยมใช้งานซึ่งจำแนกออกได้เป็น 3 ตระกูลหลัก ๆ คือ เชลล์แบบบอร์น (Bourne Shell) เชลล์แบบซี (C Shell) และเชลล์แบบคอร์น (Korn Shell) เชลล์ที่ได้รับความนิยมใช้งานมากที่สุดก็คือเชลล์แบบบอร์นและเชลล์แบบซี โดยปกติระบบปฏิบัติการ UNIX มักจะรวมเชลล์ทั้ง 3 ชนิดไว้เป็นส่วนหนึ่งของระบบปฏิบัติการตั้งแต่แรกเริ่มติดตั้ง ทั้งนี้ผู้ใช้งานสามารถเลือกได้ว่าในขณะเวลาหนึ่ง ๆ ตนต้องการจะใช้เชลล์ชนิดใด

เชลล์แต่ละชนิดต่างมีการรวมชุดคำสั่งใช้งานและยูทิลิตี้ต่าง ๆ ซึ่งมีประสิทธิภาพในการทำงานอย่างสูงอยู่เป็นจำนวนมาก ผู้ใช้งานอาจเลือกใช้คำสั่งใดคำสั่งหนึ่งซึ่งให้ผลสอดคล้องกับความต้องการของตนได้ แต่ในบางกรณีที่ความต้องการของผู้ใช้งานค่อนข้างซับซ้อน อาจจำเป็นต้องมีการเรียกใช้คำสั่งหลาย ๆ ชุดเพื่อให้ผลของการปฏิบัติงานเป็นไปตามต้องการ ที่สำคัญก็คือมีงานอีกจำนวนหนึ่งซึ่งไม่สะดวกที่จะป้อนคำสั่งโดยลำดับได้

บางครั้งอาจต้องมีการนำผลที่ได้จากการใช้คำสั่งหนึ่งไปเป็นข้อมูลให้กับอีกคำสั่งหนึ่ง รวมถึงการสร้างและกำหนดเงื่อนไขต่าง ๆ เพื่อให้สอดคล้องกับความต้องการของผู้ใช้งาน ในกรณีนี้จะเห็นว่ารูปแบบการแก้ไขปัญหาโดยเขียนโปรแกรมใช้งานจะสามารถให้ผลในการปฏิบัติงานที่ดีกว่า ระบบปฏิบัติการ UNIX เป็นระบบปฏิบัติการที่ผู้ใช้งานสามารถทำการสร้างโปรแกรมใช้งานขึ้นโดยการนำเอาชุดคำสั่งต่าง ๆ ที่มีอยู่ในเชลล์มาเขียนเป็นโปรแกรมร่วมกับฟังก์ชันพิเศษ สัญลักษณ์สนับสนุน และเงื่อนไข (condition) ต่าง ๆ ที่มีการจัดเตรียมไว้ โดยผู้ใช้งานสามารถควบคุมการเขียนโปรแกรมผ่านตัวแปร (Variable) ซึ่งมีอยู่หลายชนิดด้วยกัน

ผู้เขียนขอเริ่มบทความด้วยการเขียนโปรแกรมบนเชลล์แบบบอร์น แล้วจึงจะได้กล่าวถึงกรณีของเชลล์แบบซีต่อไป ขั้นแรกให้ทดลองสร้างโปรแกรมเชลล์หรือที่เรียกกันทั่วไปว่า เชลล์สคริปต์ (Shell Script) ง่าย ๆ ขึ้นมาสักหนึ่งโปรแกรม

ทดลองสร้างเชลล์สคริปต์

ผู้เขียนขอยกตัวอย่างเชลล์สคริปต์ชื่อ **checkuser** ดังมีรายละเอียดตามที่แสดงในกรอบที่ 1 ซึ่งเป็นการรวมคำสั่ง **date, echo** และ **who** โดยผลที่ได้จากการสั่งทำงานก็คือเชลล์จะทำการแสดงวันและเวลาปัจจุบันพร้อมกับรายชื่อของผู้ที่กำลังใช้งานระบบปฏิบัติการอยู่ หากเป็นไปได้ก็ขอให้ผู้อ่านทำการสร้างโปรแกรมตามไปด้วย ทั้งนี้อาจใช้เอดิเตอร์ **vi** หรือใช้คำสั่ง **cat > checkuser** ซึ่งในกรณีหลังนี้เมื่อพิมพ์ถึงบรรทัดสุดท้าย (who) แล้วก็ให้กดปุ่ม CTRL และ D พร้อม ๆ กัน เพื่อเป็นการเขียนไฟล์ลงบนฮาร์ดดิสก์

จากกรอบที่ 1 จะเห็นว่าเมื่อทำการสั่งให้โปรแกรม **checkuser** ทำงานโดยการพิมพ์ชื่อโปรแกรมตามหลังเครื่องหมาย $ (เครื่องหมายพรอมต์ของเชลล์แบบบอร์น) เชลล์จะตอบปฏิเสธกลับมาว่าผู้เขียนไม่มีสิทธิในการสั่งใช้งานไฟล์ดังกล่าว ทั้งนี้เพราะไฟล์ที่ถูกสร้างขึ้นใหม่ในระบบปฏิบัติการ UNIX ไม่ว่าจะโดยเอดิเตอร์ตัวใดก็ตามจะได้รับการกำหนดสถานะการใช้งานเป็น rw-r—r— เสมอ ซึ่งอ่านค่าได้ว่าเจ้าของไฟล์ (ก็คือผู้สร้างไฟล์นี้ขึ้นมา) สามารถทำได้เพียงอ่านหรือแก้ไขเนื้อหาภายในไฟล์เท่านั้น ผู้ใช้งานรายอื่นไม่ว่าจะอยู่ในกลุ่มผู้ใช้งานเดียวกันหรือไม่ต่างก็ทำได้เพียงแค่อ่านไฟล์เท่านั้น จึงเป็นเรื่องควรจำว่าทุกครั้งที่สร้างไฟล์เชลล์สคริปต์ขึ้นใหม่ ผู้สร้างจะต้องทำการกำหนดสิทธิในการใช้งานให้กับไฟล์ด้วยเสมอ ในตัวอย่างนี้ผู้เขียนเป็นผู้สร้างไฟล์ และต้องการสั่งใช้งานไฟล์นี้เพียงคนเดียว จึงใช้คำสั่ง **chmod** เพื่อกำหนดสิทธิในการใช้งานไฟล์ **checkuser** เป็น rwxr—r— รายละเอียดของสิทธิในการใช้งานไฟล์ก่อนและหลังทำการแก้ไขได้แสดงไว้ในกรอบที่ 2 สำหรับรายละเอียดในเรื่องของการอ่านค่าสิทธิในการใช้ไฟล์ รวมถึงคำสั่งต่าง ๆ ที่ใช้ในการแก้ไขสิทธิ ขอให้ลองหาอ่านจากบทความชุด "ก้าวแรก... สู่โลก UNIX"

กรอบที่ 1

```
$ cat checkuser
date
echo Listing of user currently logged in...
who
$ checkuser
checkuser: execute permission denies
```

กรอบที่ 2

```
$ ls -lg checkuser
-rw-r—r— 1   pairoj   mgt   48 Jun 21   10:42   checkuser
$ chmod 744 checkuser                    ** หรือ chmod u+x checkuser **
$ ls -lg checkuser
-rwxr—r— 1   pairoj   mgt   48 Jun 21   10:42   checkuser
```

คราวนี้ลองมาดูผลที่ได้จากการสั่งใช้งานโปรแกรม **checkuser** ตามที่แสดงในกรอบที่ 3 ในบางครั้งผลที่ได้อาจมีรายละเอียดปลีกย่อยแตกต่างไปจากที่แสดงในตัวอย่าง ทั้งนี้ขึ้นอยู่กับตระกูลของระบบปฏิบัติการ UNIX ที่ผู้ใช้งานทดลองใช้อยู่ ในกรณีที่ต้องการกำหนดสิทธิในการใช้งานโปรแกรมดังกล่าวให้กับผู้ใช้งานรายอื่นซึ่งอยู่ในกลุ่มผู้ใช้งานเดียวกันก็ทำได้โดยการใช้คำสั่ง **chmod 774 checkuser** หากผู้ใช้งานไม่สามารถเรียกใช้โปรแกรมได้ทั้ง ๆ ที่สิทธิในการใช้งานไฟล์ถูกต้องและไดเรคทอรีใช้งานปัจจุบันก็เป็นไดเรคทอรีเดียวกับไฟล์แล้ว ซึ่งโดยส่วนใหญ่จะได้รับข้อความตอบปฏิเสธกลับมาจากเชลล์ว่า "checkuser: Command not found" ก็อาจเป็นได้ว่าเงื่อนไขตัวแปรในขณะทำการ login ของผู้อ่านมิได้กำหนดให้มีการค้นหาไฟล์ใช้งานจากไดเรคทอรีปัจจุบัน ในกรณีนี้ขอให้ผู้ใช้งานเรียกใช้โปรแกรมโดยการพิมพ์ **./checkuser** ซึ่งจะเป็นการบอกให้เชลล์ทำการค้นหาไฟล์จากไดเรคทอรีใช้งานปัจจุบัน หรือมิฉะนั้นเพื่อเป็นการสะดวกขึ้นก็ขอให้ลองทำการแก้ไขตัวแปร **PATH** เพื่อให้สามารถเรียกใช้งานโปรแกรมที่อยู่ภายในไดเรคทอรีใช้งานปัจจุบันได้ตลอดเวลา รายละเอียดติดตามได้ในบทความตอนต่อไป

กรอบที่ 3

```
$ checkuser
Sun Jun 21 10:43:27 GMT 1998
Listing of user currently logged in...
pairoj      console   Jun 21   09:57
apichart    tty03     Jun 21   10:15
wallaya     tty04     Jun 21   08:04
```

ตัวอย่างที่ผู้เขียนได้ยกขึ้นมาข้างต้นเป็นเพียงการเขียนเชลล์สคริปต์อย่างง่าย ๆ ไม่สลับซับซ้อนอะไร ซึ่งผู้อ่านจะเห็นว่าก็เป็นแค่การเรียงลำดับชุดคำสั่งของ UNIX เข้าด้วยกันเท่านั้น ก่อนที่จะทำการกล่าวถึงรายละเอียดที่ซับซ้อนขึ้นไปกว่านี้ ผู้เขียนขอกล่าวอธิบายถึงเครื่องหมายพิเศษต่าง ๆ ที่จะใช้เป็นเครื่องมือสำคัญสำหรับควบคุมการทำงานของเชลล์สคริปต์ ซึ่งก็ได้แก่เครื่องหมาย ; \ | และ & บรรดาเครื่องหมายเหล่านี้จะมีปรากฏอยู่เสมอในไฟล์เชลล์สคริปต์ทั่ว ๆ ไป

เครื่องหมาย ; (อ่านว่าเซมิโคลอน - semicolon) เป็นเครื่องหมายที่ใช้แยกชุดคำสั่งออกจากกัน โดยลำดับการทำงานของชุดคำสั่งจะทำเรียงจากซ้ายไปขวาทีละคำสั่ง ตัวอย่างเช่น **date ; who ; cal** จะมีลำดับการทำงานโดยเริ่มจากการแสดงวันและเวลาปัจจุบัน จากนั้นจึงแสดงรายชื่อของผู้ที่กำลัง login ระบบปฏิบัติการ UNIX อยู่ แล้วจึงแสดงปฏิทินของปีปัจจุบัน ผู้ใช้งานสามารถเขียนแยกระยะห่างระหว่างคำสั่งแต่ละคำสั่งกับเครื่องหมาย ; เท่าใดก็ได้ไม่มีผลกระทบต่อการทำงานแต่อย่างใด หลาย ๆ ท่านคงคิดว่าก็ไม่ต่างอะไรกับการเขียนคำสั่งแยกกันคนละบรรทัด ซึ่งจริง ๆ ก็เป็นเช่นนั้น เครื่องหมายดังกล่าวมักพบใช้ในบางกรณีที่ต้องการรวมการทำงานกลุ่มเดียวกันไว้บรรทัดเดียวกันเพื่อความสะดวกในการอ่านของผู้เขียนสคริปต์บางรายเท่านั้นเอง ส่วนเครื่องหมาย \ จะใช้ในกรณีที่ไฟล์เชลล์สคริปต์มีบรรทัดใดบรรทัดหนึ่งยาวมากจนตกขอบจอของเอดิเตอร์ ทำให้ไม่สะดวกสำหรับการตรวจแก้ไขในภายหลัง จึงมีการออกแบบให้ใช้เครื่องหมาย \ เพื่อเชื่อมข้อความบรรทัดที่อยู่ติดกันซึ่งเป็นชุดคำสั่งเดียวกันให้เชลล์รับทราบ ในกรณีนี้ผู้เขียนสคริปต์สามารถเคาะปุ่ม ENTER เพื่อเริ่มข้อความส่วนที่เหลือในบรรทัดใหม่ได้โดยปิดบรรทัดก่อนหน้าด้วยเครื่องหมาย \

เครื่องหมาย | (อ่านว่าไปป์ - pipe) ใช้คั่นระหว่างคำสั่งสองคำสั่งในกรณีที่ผู้เขียนสคริปต์ต้องการนำผลลัพธ์หรือเอาต์พุตที่ได้จากคำสั่งแรกส่งไปเป็นอินพุตของคำสั่งที่สอง ตัวอย่างที่ขอยกขึ้นมาก็คือ **who | grep pairoj | lpr** อธิบายได้ดังนี้คือ คำสั่ง **who** จะส่งรายชื่อของผู้ที่กำลังทำการ login ต่อไปเป็นอินพุตให้กับคำสั่ง **grep** ซึ่งกำหนดเงื่อนไขให้เลือกแสดงเฉพาะบรรทัดที่มีคำว่า pairoj อยู่ ผลลัพธ์ที่ได้จากคำสั่ง **grep** จะถูกส่งต่อไปเป็นอินพุตให้กับคำสั่ง **lpr** ซึ่งทำการพิมพ์ข้อมูลที่ได้รับออกทางเครื่องพิมพ์ ความสามารถดังกล่าวเรียกว่าการไปป์ ความสามารถพิเศษเหล่านี้ส่งผลให้ผู้ใช้งานระบบปฏิบัติการ UNIX สามารถสร้างชุดคำสั่งพิเศษใหม่ ๆ ออกมาใช้งานตามวัตถุประสงค์ของตนเองโดยอยู่ในรูปของเชลล์สคริปต์ได้อย่างไม่มีขีดจำกัด ซึ่งเมื่อรวมกับชุดคำสั่งเงื่อนไขที่จะกล่าวถึงต่อไปแล้วก็ยิ่งทำให้ UNIX เป็นระบบปฏิบัติการที่มีความยืดหยุ่นในการนำไปประยุกต์ใช้งานสูงมากทีเดียว

ในเชลล์สคริปต์บางโปรแกรม ผู้ออกแบบมีความจำเป็นที่จะต้องกำหนดให้บางคำสั่งทำงานแบบหลังฉาก (Background task) โดยใช้เครื่องหมาย & ตามหลังคำสั่งที่ต้องการให้มีการทำงานแบบหลังฉาก การอธิบายความหมายของการทำงานแบบหลังฉากคงจะไม่มีวิธีใดดีไปกว่าการยกตัวอย่างจริง ผู้เขียนจะลองเปรียบเทียบความแตกต่างของการทำงานภายในเชลล์สคริปต์ 2 ไฟล์ดังแสดงในกรอบที่ 4 ซึ่งเป็นการอ้างชุดคำสั่งที่ใช้งานบนระบบปฏิบัติการ Solaris อันเป็น UNIX ตระกูลหนึ่งที่ได้รับความนิยมใช้งานอย่างมาก

กรอบที่ 4

เชลล์สคริปต์ ก.

```
...
cp login1 login.sts
xclock -analog
calc
...
```

เชลล์สคริปต์ ข.

```
...
cp login1 login.sts
xclock -analog &
calc &
...
```

จากตัวอย่างเป็นการเรียกชุดคำสั่งใช้งานในโหมดการทำงานแบบกราฟิก เริ่มจากการสำเนาไฟล์ login1 ไปเป็นไฟล์ชื่อ login.sts จากนั้นจึงเป็นการสั่งให้แสดงนาฬิกาบอกเวลาแบบอะนาล็อก แล้วจึงเรียกใช้เครื่องคิดเลข ข้อผิดพลาดที่เกิดขึ้นในเชลล์สคริปต์ ก. ก็คือเชลล์จะไม่สามารถเรียกคำสั่ง **calc** ได้เลย ทั้งนี้เพราะรูปแบบการเขียนเชลล์สคริปต์ซึ่งได้กล่าวมาตั้งแต่แรกนั้นจะมีลำดับการทำงานเรียงตามบรรทัด เชลล์จะรอจนกว่าคำสั่งหนึ่งจะทำงานเสร็จสิ้นแล้วจึงเลื่อนไปทำงานในคำสั่งถัดไป สำหรับตัวอย่าง ก. เชลล์จะรอจนกว่าคำสั่ง **xclock** สิ้นสุดการทำงาน (เลิกการทำงาน) ซึ่งผู้อ่านคงเห็นได้ว่าไม่มีทางเป็นไปได้หากไม่มีการสั่งยกเลิกจากภายนอก ในกรณีนี้ควรเปลี่ยนรูปแบบการเรียกใช้คำสั่งดังกล่าวไปให้ทำงานเป็นแบบหลังฉากโดยการเพิ่มเครื่องหมาย & ตามหลังคำสั่ง **xclock** จะทำให้เชลล์ไม่รอให้คำสั่งดังกล่าวสิ้นสุดการทำงาน แต่จะเลื่อนไปปฏิบัติตามคำสั่งถัดไปในทันทีที่สามารถเรียกใช้คำสั่ง **xclock** ได้สำเร็จ จะเห็นว่าในบรรทัดถัดไปก็จำเป็นต้องกำหนดให้ทำงานแบบหลังฉากด้วยเช่นกัน อนึ่ง แต่ละคำสั่งที่ถูกกำหนดให้ทำงานแบบหลังฉากจะได้รับการแจ้งเลขหมายแสดงโปรเซส (Process Identification : PID) จากเชลล์ใช้งาน ทั้งนี้เพื่อเป็นประโยชน์ในการอ้างถึงในภายหลังไม่ว่าจะโดยตัวเชลล์สคริปต์เองหรือโดยผู้ใช้งานรายอื่น ๆ

แถมท้ายด้วยเรื่องของการจัดกลุ่มคำสั่ง ซึ่งพบได้ในบางครั้งที่ผู้ออกแบบเชลล์ต้องการรวมคำสั่งมากกว่าหนึ่งชุดให้ทำงานภายใต้เงื่อนไขเดียวกัน ทั้งนี้ทำได้ด้วยการรวมกลุ่มคำสั่งไว้ภายในเครื่องหมาย ( ) ซึ่งเชลล์จะทำการสร้างเชลล์ใหม่ขึ้นมาเพื่อรองรับการทำงานกับคำสั่งภายในเครื่องหมายวงเล็บ เชลล์ที่ถูกสร้างขึ้นใหม่นี้มีชื่อเรียกว่าซับเชลล์ (subshell) ตัวอย่างเช่น **(a ; b) & c** ซึ่งหมายถึงการจัดให้คำสั่ง **a** และ **b** ทำงานแบบหน้าฉาก ส่วนคำสั่ง **c** ให้ทำงานแบบหลังฉาก ที่กล่าวถึงก็เพราะว่าในบางครั้งผู้อ่านอาจพบการจัดรูปแบบคำสั่งในลักษณะนี้

การเปลี่ยนทิศทางข้อความแสดงความผิดพลาด

ความสามารถประการหนึ่งของระบบปฏิบัติการ UNIX ก็คือการอนุญาตให้ผู้ใช้งานทำการเปลี่ยนทิศทางการแสดงผลของข้อมูล โดยปกติเอาต์พุตมาตรฐานของ UNIX ก็คือหน้าจอเทอร์มินัลใช้งานของผู้ใช้ ผู้ใช้งานสามารถปรับทิศทางการแสดงผลจากที่ส่งมายังหน้าจอให้ไปยังอุปกรณ์เอาต์พุตอื่น ๆ เช่น อาจเป็นการเขียนลงในไฟล์ข้อความ หรือส่งไปยังเครื่องพิมพ์ เป็นต้น ความสามารถดังกล่าวมีชื่อเรียกเป็นทางการว่า การเปลี่ยนทิศทาง (Redirection) ผู้ออกแบบเชลล์สคริปต์สามารถนำเอาความสามารถนี้ไปประยุกต์ใช้สำหรับส่งข้อความแจ้งเตือนหรือข้อความแสดงความผิดพลาดออกจากภาคแสดงผลมาตรฐาน เรื่องนี้มีความสำคัญมากในกรณีที่ไม่ต้องการให้ข้อความแจ้งความผิดพลาด

ต่าง ๆ ถูกแสดงปนออกมากับข้อความที่ต้องการแสดงเพื่อติดต่อกับผู้ใช้งานตามปกติ โปรแกรมประยุกต์ที่ได้รับการเขียนขึ้นต่างก็ต้องอาศัยความสามารถนี้ในการแยกแยะเนื้อความที่ได้รับจากการทำงานของเชลล์เพื่อให้ผู้ใช้งานไม่เกิดความสับสน ผู้เขียนขอยกตัวอย่างการใช้ความสามารถดังกล่าวประกอบการอธิบายเพื่อให้เข้าใจได้มากขึ้นดังนี้

ผู้เขียนขอยกตัวอย่างการใช้คำสั่ง **cat** ซึ่งเป็นคำสั่งให้เชลล์ทำการพิมพ์ข้อความซึ่งอยู่ในไฟล์ที่อ้างถึง ในกรณีที่อ้างชื่อไฟล์ซึ่งไม่มีอยู่จริง เชลล์จะแจ้งเตือนกลับมา จากกรอบที่ 5 ซึ่งมิได้ทำการเปลี่ยนทิศทางข้อความใด ๆ จะเห็นว่าเมื่อผู้เขียนป้อนคำสั่ง **cat y** และ **cat x y** โดยที่ในไดเรคทอรีใช้งานปัจจุบันมีเฉพาะไฟล์ y เท่านั้น เชลล์จะทำการแสดงเนื้อหาภายในไฟล์ y ออกมา แต่จะแจ้งความผิดพลาดขึ้นในกรณีของไฟล์ x

กรอบที่ 5

```
$ cat y
This is y.
$ cat x y
cat: x: No such file or directory
This is y.
```

คราวนี้ลองใช้ความสามารถในการเปลี่ยนทิศทางข้อมูลโดยใช้เครื่องหมาย > ทำการย้ายทิศทางการแสดงข้อความปกติไปยังไฟล์ชื่อ hold แทนที่จะเป็นภาคแสดงผลปกติตามตัวอย่างในกรอบที่ 6 จะเห็นว่าข้อความแสดงผลของคำสั่ง **cat x y** จะไม่แสดงเนื้อหาของไฟล์ y ให้ปรากฏขึ้นบนจอภาพ แต่จะถูกเขียนลงในไฟล์ชื่อ hold แทน สำหรับข้อความผิดพลาดอันเกิดจากคำสั่งการอ่านไฟล์ x ก็จะยังคงถูกแสดงอยู่บนหน้าจอตามเดิม ปรากฏการณ์นี้เป็นจริงเช่นเดียวกันในกรณีที่ใช้คำสั่งไปป์ (pipe) ตามตัวอย่างในกรอบที่ 6 อันเป็นการสั่งให้นำผลที่ได้จากการใช้คำสั่ง **cat x y** ไปเป็นอินพุตของคำสั่ง **tr "[a-z]" "[A-Z]"** ซึ่งจะทำการแปลงตัวอักษรที่ได้รับมาจากอักษรตัวพิมพ์เล็กไปเป็นอักษรตัวพิมพ์ใหญ่ จากตัวอย่างนี้ก็จะเห็นว่าข้อความแสดงความผิดพลาดยังคงปรากฏอยู่บนหน้าจอเช่นเดิม สรุปก็คือการเปลี่ยนทิศทางโดยทั่วไปรวมถึงการไปป์ข้อมูลไม่มีผลกระทบแต่อย่างใดต่อข้อความแสดงความผิดพลาด ดังนั้นหากผู้ออกแบบโปรแกรมเชลล์สคริปต์ต้องการเปลี่ยนทิศทางการแสดงความผิดพลาดให้ไปยังอุปกรณ์อื่นก็จำเป็นจะต้องใช้เทคนิคการเปลี่ยนทิศทางแบบพิเศษดังจะกล่าวถึงต่อไปนี้

กรอบที่ 6

```
$ cat x y > hold
cat: x: No such file or directory
$ cat hold
This is y.

$ cat x y | tr "[a-z]" "[A-Z]"
cat: x: No such file or directory
THIS IS Y.
```

จากตัวอย่างในกรอบที่ 7 มีการเปลี่ยนแปลงรูปแบบของเครื่องหมายเปลี่ยนทิศทาง โดยแบ่งเป็นเครื่องหมาย 1> ซึ่งใช้สำหรับเปลี่ยนทิศทางของข้อความปกติและ 2> ซึ่งใช้สำหรับเปลี่ยนทิศทางของข้อความแสดงความผิดพลาด จะเห็นว่าเครื่องหมาย 1> มีความหมายเทียบเท่ากับเครื่องหมาย > ที่ได้กล่าวถึงในตัวอย่างแรก ๆ นั่นเอง จากตัวอย่างจะเห็นว่าข้อความปกติถูกส่งไปเขียนลงในไฟล์ที่ชื่อ hold1 และข้อความแสดงความผิดพลาดถูกเขียนลงในไฟล์ชื่อ hold2 ในกรณีที่ต้องการเขียนข้อความแสดงความผิดพลาดนำหน้าข้อความปกติลงในไฟล์เดียวกันก็สามารถทำได้โดยเปลี่ยนเครื่องหมายเปลี่ยนทิศทางจาก 2> hold2 ไปเป็น 2>&1 ดังตัวอย่างแสดงในกรอบที่ 7 เช่นเดียวกัน

กรอบที่ 7

ต้องการแยกผลออกจากกัน

```
$ cat x y 1> hold1 2> hold2
$ cat hold1
This is y.
$ cat hold2
cat: x: No such file or directory
```

ต้องการรวมผลเข้าด้วยกัน

```
$ cat x y 1> hold 2>&1
$ cat hold
cat: x: No such file or directory
This is y.
```

ผู้ออกแบบโปรแกรมสามารถแก้ไขเครื่องหมายเปลี่ยนทิศทางจาก 2>&1 ไปเป็น 1>&2 ซึ่งจะเป็นการนำผลที่ได้จากการแสดงข้อความปกติไปรวมอยู่ในไฟล์ของการแสดงข้อความผิดพลาด เทคนิคเหล่านี้ได้รับการหยิบยกนำไปใช้เสมอสำหรับการออกแบบโปรแกรมเชลล์สคริปต์ ซึ่งผู้ออกแบบต้องการส่งข้อความแสดงความผิดพลาดไปยังไฟล์พิเศษที่ต้องการแทนที่จะให้ส่งออกมาปรากฏบนหน้าจอ

เรื่องของโปรเซส

ก่อนที่จะกล่าวถึงเรื่องของการเขียนโปรแกรมเชลล์สคริปต์กันต่อไป ผู้เขียนขอขัดจังหวะด้วยการนำเสนอเรื่องราวของโปรเซส (process) ซึ่งมีการกล่าวถึงไปบ้างแล้วในช่วงต้น ๆ ของบทความตอนนี้ การใช้งานระบบปฏิบัติการ UNIX รวมถึงการเขียนโปรแกรมเชลล์สคริปต์โดยไม่ศึกษาถึงความหมายของโปรเซสเลยนั้นก็เปรียบเสมือนการขับรถโดยไม่มีความรู้ในเรื่องการทำงานของเครื่องยนต์เลย ในส่วนสุดท้ายของบทความชุดนี้ ผู้เขียนจึงขอกล่าวถึงเรื่องของโปรเซส

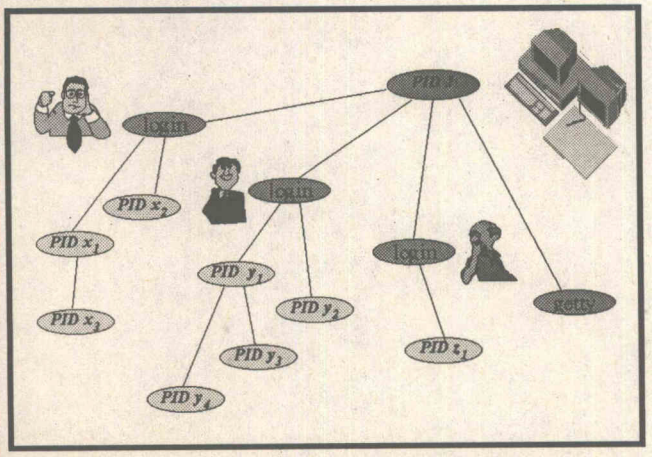
โปรเซสเป็นคำที่ใช้เรียกการทำงานของระบบปฏิบัติการ UNIX ซึ่งเกิดขึ้นเมื่อเชลล์ได้รับคำสั่งจากผู้ใช้งาน โดยที่เชลล์จะทำการสร้างโปรเซสขึ้นหนึ่งโปรเซสเพื่อทำงานตามคำสั่งที่ได้รับมาหนึ่งคำสั่ง ในกรณีของเชลล์สคริปต์อาจเป็นไปได้ที่มีการเรียกใช้คำสั่งขึ้นพร้อม ๆ กันหลายคำสั่งซึ่งทำให้เกิดโปรเซสขึ้นพร้อม ๆ กันหลายโปรเซส เมื่อมอง

ในมุมมองของการเป็นระบบปฏิบัติการซึ่งสามารถรองรับการใช้งานจากหลายผู้ใช้งานจะพบว่ามีโปรเซสจำนวนมากซึ่งเกิดขึ้นเพื่อตอบรับการเรียกใช้งานของผู้ใช้งานแต่ละรายในเวลาเดียวกัน สิ่งสำคัญซึ่งทั้งผู้ใช้งานระบบปฏิบัติการ UNIX และผู้ออกแบบโปรแกรมเชลล์สคริปต์จะต้องทราบก็คือความสัมพันธ์ของโปรเซสแต่ละโปรเซสและการควบคุมโปรเซสของ UNIX เอง การทำความเข้าใจโครงสร้างการจัดการโปรเซสในระบบปฏิบัติการ UNIX ก็คงจะไม่มีวิธีใดดีไปกว่าการเทียบกับโครงสร้างการจัดการของระบบไฟล์ทั่วไปซึ่งมีลักษณะเป็นแผนแบบลำดับขั้น (Hierachical Diagram) ความสัมพันธ์ระหว่างโปรเซสจะอยู่ในลักษณะของพ่อกับลูก (parent and child) โดยที่โปรเซสในระดับพ่อสามารถสร้างโปรเซสในระดับลูกจำนวนเท่าใดก็ได้ ระบบปฏิบัติการ UNIX จะใช้คำว่าฟอล์ค (Folk) หรือสปอร์น (spawn) เรียกแทนการสร้างโปรเซสในระดับลูกให้เกิดขึ้น ทั้งนี้ขอให้ศึกษาจากรูปที่ 1 ประกอบไปด้วย ในระบบไฟล์ ไดเรคทอรีสูงสุดก็คือรากหรือรูต (root) แต่สำหรับในเรื่องของระบบโปรเซสจะเป็นโปรเซสหมายเลข 1 (PID 1)

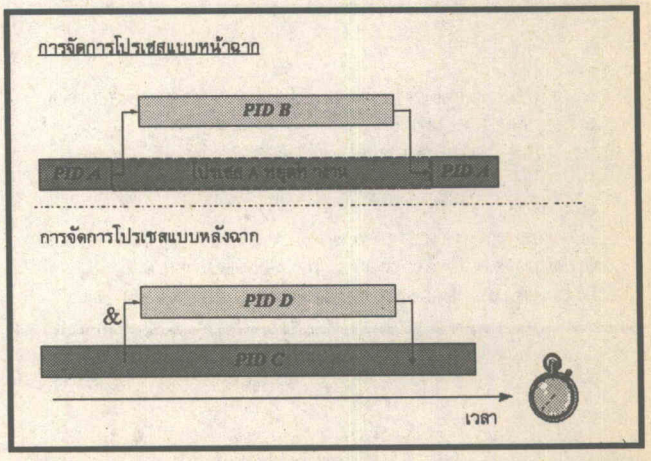
เริ่มจากเมื่อมีการเปิดเครื่องคอมพิวเตอร์ หลังจากที่ผ่านกระบวนการเริ่มต้นทางด้านซอฟต์แวร์ทั้งหมดแล้ว ระบบปฏิบัติการ UNIX จะสร้างโปรเซสเริ่มต้นขึ้นเป็นโปรเซสหมายเลข 1 ซึ่งถือเป็นโปรเซสเริ่มต้นที่จะสร้างโปรเซสในระดับลูกทั้งหลายต่อไป เทียบได้กับไดเรคทอรีรากในระบบไฟล์ทั่วไป ไม่ว่าจะมีการเชื่อมต่อเทอร์มินัลสำหรับให้ผู้ใช้รายอื่นสามารถติดต่อผ่านเครื่องคอมพิวเตอร์ หรือแม้จะมีเพียงเฉพาะคอนโซลใช้งานของตัวเครื่องคอมพิวเตอร์นั้นเองก็ตาม โปรเซสหมายเลข 1 จะดำเนินการสร้างโปรเซสที่มีชื่อว่า **getty** ให้กับแต่ละเทอร์มินัลเพื่อคอยดักจับพฤติกรรมการใช้งานของผู้ใช้งาน โดยปกติโปรเซส **getty** จะไม่มีการทำงานใด ๆ ต่อเมื่อผู้ใช้งานทำการ login ผ่านเข้ามาทางเทอร์มินัลใด โปรเซส getty ของเทอร์มินัลดังกล่าวก็จะเริ่มต้นทำงานโดยเปลี่ยนการทำงานไปเป็นโปรเซส **login** เพื่อโต้ตอบกับผู้ใช้งานในช่วงของการ login และในที่สุดก็จะเปลี่ยนไปเป็นโปรเซสของผู้ใช้งานตามปกติ เมื่อผู้ใช้งานทำการ logout ออกไปแล้ว โปรเซสดังกล่าวก็จะเปลี่ยนกลับมาเป็น **getty** เพื่อรอรับการ login ครั้งต่อไป

เมื่อผู้ใช้งานรายใดรายหนึ่งได้ผ่านกระบวนการ login และอยู่ในสภาพพร้อมใช้งานแล้ว หากมีการเรียกใช้คำสั่งใด ๆ เกิดขึ้น โปรเซสที่รอรับคำสั่งจะทำการสร้างโปรเซสลูกขึ้นเพื่อปฏิบัติตามคำสั่งที่ได้รับมา ในขณะที่โปรเซสลูกกำลังทำงานที่ได้รับมอบหมายนั้น โปรเซสพ่อจะหยุดทำงานชั่วคราวโดยไม่รับรู้การป้อนคำสั่งใด ๆ จากผู้ใช้งานและไม่สามารถติดต่อกับอุปกรณ์ต่าง ๆ ในเครื่องคอมพิวเตอร์ (เรียกว่าภาวะหลับหรือ sleep) สิ่งที่ทำก็เพียงรอให้โปรเซสลูกทำงานเสร็จสิ้นเท่านั้น เมื่อโปรเซสลูกปฏิบัติงานตามคำสั่งเสร็จสิ้นแล้วก็จะสลายตัวไป อันจะทำให้โปรเซสพ่อตื่นขึ้น (wake-up) เพื่อให้บริการผู้ใช้งานต่อไป ตัวอย่างง่าย ๆ ที่เห็นได้ก็คือการใช้คำสั่ง **find** ในการค้นหาไฟล์ที่ต้องการในฮาร์ดดิสก์ ซึ่งผู้ใช้งานจะไม่สามารถป้อนคำสั่งเพื่อให้เชลล์ทำงานตามความต้องการได้ในทันที แต่จะต้องรอจนกว่าโปรเซสที่เกิดขึ้นเพื่อรองรับคำสั่ง **find** จะทำงานเสร็จสิ้นเสียก่อน เรียกการทำงานแบบนี้ว่าการจัดการโปรเซสแบบหน้าฉาก (Foreground Process)

ในกรณีที่ผู้ใช้งานหรือผู้เขียนโปรแกรมเชลล์สคริปต์ไม่ต้องการให้โปรเซสพ่อหยุดรอการทำงานจากโปรเซสลูก ก็สามารถทำได้โดยการบังคับให้โปรเซสลูกทำงานแบบหลังฉาก (Background Process) ซึ่งทำได้ง่าย ๆ โดยการเติมเครื่องหมาย & ตามหลังคำสั่งที่จะให้โปรเซสลูกจัดการ เช่น **find / -name "core" -ok rm {} \; &** เป็นต้น ในกรณีนี้เชลล์จะทำการสร้างโปรเซสลูกขึ้นโดยที่โปรเซสพ่อจะไม่หยุดทำงานและไม่รอว่าโปรเซสลูกจะทำงานเสร็จเมื่อใด สิ่งที่เกิดขึ้นก็มีเพียงโปรเซสพ่อทำการสร้างโปรเซสลูกขึ้น แจ้งเลขหมายประจำโปรเซส (PID) ของโปรเซสลูกแล้วจึงกลับมาอยู่ในสภาวะพร้อมรับคำสั่งต่อไป สำหรับโปรเซสลูกก็จะดำเนินการตามคำสั่งที่ได้รับโดยอิสระ ไม่สนใจโปรเซสพ่อแต่อย่างใดเช่นกัน ลองเปรียบเทียบแผนภาพแสดงการจัดการโปรเซสแบบหน้าฉากและหลังฉากจากรูปที่ 2 จะเข้าใจได้มากขึ้น



รูปที่ 1 โครงสร้างของการจัดการโปรเซสในระบบปฏิบัติการ UNIX



รูปที่ 2 เปรียบเทียบความแตกต่างระหว่างโปรเซสหน้าฉากและหลังฉาก

เรื่องสำคัญอีกประการหนึ่งก็คือกระบวนการในการจัดสรรหมายเลขประจำโปรเซส สมมติว่าผู้ใช้งานได้รับการให้บริการโดยโปรเซสหนึ่ง ๆ อยู่ (โดยทั่วไปก็น่าจะเป็นโปรเซส login แต่ผู้ใช้งานอาจสร้างเชลล์เพิ่มก็ได้ จะขอกล่าวถึงในภายหลัง) โปรเซสดังกล่าวจะมีหมายเลขประจำ (PID) กำกับอยู่ เลขหมายดังกล่าวก็จะคงอยู่กับโปรเซสนี้ไปเรื่อย ๆ จนกว่าผู้ใช้งานจะสั่งยุติการทำงานของโปรเซสนี้ไป ตัวอย่างเช่น ทำการ logout ออกจากระบบปฏิบัติการ เป็นต้น เมื่อใดก็ตามที่ผู้ใช้งานมีการเรียกใช้คำสั่งใด ๆ ขึ้น โปรเซสนี้ก็จะทำการสร้างโปรเซสลูกขึ้น ไม่ว่าโปรเซสลูกจะทำงานแบบหน้าฉากหรือหลังฉากก็ตาม โปรเซสลูกก็จำเป็นต้องได้รับการกำหนดหมายเลขประจำตัวด้วยเช่นเดียวกัน แต่ทั้งนี้มิได้เป็นกฎตายตัวว่าเมื่อผู้ใช้งานสั่งคำสั่งเดิมซ้ำอีกครั้ง โปรเซสลูกที่เกิดขึ้นใหม่ในครั้งที่สองจะมีหมายเลขประจำโปรเซสเดียวกันกับในครั้งแรก

ผู้เขียนขอยกตัวอย่างประกอบการอธิบายตามกรอบที่ 8 โดยใช้คำสั่ง **ps** ซึ่งใช้แสดงรายละเอียดต่าง ๆ ของโปรเซสที่เกิดขึ้นอยู่ในระบบปฏิบัติการ UNIX ผู้เขียนลองเรียกใช้คำสั่งดังกล่าวซ้ำเดิมจำนวน 2 ครั้ง ขอให้ลองดูผลที่ได้รับกลับโดยดูจากคอลัมน์ชื่อ PID (Process Identification) ซึ่งแสดงหมายเลขโปรเซสของคำสั่งสั่ง **ps** และ PPID (Parent's PID) ซึ่งเป็นหมายเลขโปรเซสของโปรเซสพ่อ จากการป้อนคำสั่งครั้งแรกจะเห็นในขณะนั้นมีโปรเซสเกิดขึ้นเป็นจำนวน 2 โปรเซสด้วยกันคือ โปรเซสชื่อ **sh** ซึ่งเป็นโปรเซสที่คอยรับคำสั่งจากผู้ใช้งานและถือเป็นโปรเซสพ่อของ **ps -l** สำหรับโปรเซสที่สองคือ **ps -l** ซึ่งเกิดขึ้นเพื่อตอบรับการใช้งานของผู้ใช้งาน โดย PID และ PPID ของโปรเซส **sh** มีค่าเป็น 18901 และ 18900 และสำหรับโปรเซส **ps -l** จะมีค่าเป็น 18908 และ 18901 ตามลำดับ จะเห็นจากผลที่ได้ว่าโปรเซส **ps -l** เป็นลูกของโปรเซส **sh** จากค่า PPID ซึ่งมีค่าเป็น 18901 การป้อนคำสั่ง **ps -l** ครั้งถัดไปจะพบว่าหมายเลข PID ของโปรเซส **ps -l** จะแตกต่างจากในครั้งแรก แต่ยังคงมีหมายเลข PPID ตรงกับค่าเดิมซึ่งก็แสดงว่ายังคงเป็นโปรเซสลูกของโปรเซส **sh** อยู่นั่นเอง

กรอบที่ 8

```
$ ps -l
      F  UID   PID PPID CP PRI  NI SZ RSS WCHAN STAT TT TIME COMMAND
12034102  108   18901 18900 0  15   0  72  417 kernekma  S   p2  0:00  -sh
12000002  108   18908 18901 19 29  0  212 448           R   p2  0:00  ps -l

$ ps -l
      F UID  PID  PPID CP PRI  NI SZ RSS  WCHAN STAT TT TIME COMMAND
12034102 108 18901 18900 0  15   0  72  417  kernekma S    p2  0:00  -sh
12000002 108 18914 18901 18 29  0  212  448           R    p2  0:00  ps -l
```

แถมท้ายด้วยเรื่องของการเขียนเชลล์สคริปต์อีกเล็กน้อย ผู้อ่านได้ทราบไปแล้วว่าการเขียนไฟล์เชลล์สคริปต์นั้นจะต้องทำการกำหนดสิทธิในการใช้ไฟล์ให้เป็นแบบเรียกใช้งานได้ (executable) สำหรับการเรียกใช้ก็คือพิมพ์ชื่อไฟล์ไปเลย อย่างไรก็ตาม ในกรณีที่ไม่สามารถทำการปรับเปลี่ยนสิทธิในการใช้ไฟล์ได้ อาจเป็นด้วยเพราะตนไม่ใช่ผู้เขียนไฟล์และปัจจุบันไม่มีสิทธิในการเรียกใช้งาน ก็สามารถเลี่ยงไปใช้วิธีอื่นในการเรียกใช้คำสั่งได้ โดยการใช้คำสั่ง **sh** แล้วตามด้วยชื่อไฟล์เชลล์สคริปต์ ตัวอย่างเช่น **sh checkuser** ซึ่งจะให้ผลในการทำงานเช่นเดียวกับกรณีการสั่งการตามตัวอย่างที่กล่าวถึงในตอนต้นทุกประการ

ผ่านไปแล้วสำหรับบทความในตอนแรกซึ่งถือเป็นการเตรียมพร้อมก่อนการเขียนโปรแกรมใช้งานจริง สำหรับในตอนต่อไปผู้เขียนจะขอนำเสนอเรื่องราวของตัวแปรต่าง ๆ ที่ผู้ออกแบบเชลล์สคริปต์ควรจะต้องทราบ ผู้เขียนยินดีรับฟังข้อเสนอแนะและข้อติชมผ่านทางอีเมล์โดยขอให้ส่งไปที่ wpairoj@hotmail.com

แล้วพบกันใหม่ฉบับหน้าครับ...